NAME

Math::BigFloat - Arbitrary size floating point math package

SYNOPSIS

```
use Math::BigFloat;

# Number creation
$x = Math::BigFloat->new($str); # defaults to 0
$nan  = Math::BigFloat->bnan(); # create a NotANumber
$zero = Math::BigFloat->bzero(); # create a +0
$inf = Math::BigFloat->binf(); # create a +inf
$inf = Math::BigFloat->binf('-'); # create a -inf
$one = Math::BigFloat->bone(); # create a +1
$one = Math::BigFloat->bone('-'); # create a -1

# Testing
$x->is_zero();  # true if arg is +0
$x->is_nan();   # true if arg is NaN
$x->is_one();   # true if arg is +1
$x->is_one('-');  # true if arg is -1
$x->is_odd();   # true if odd, false for even
$x->is_even();  # true if even, false for odd
$x->is_pos();   # true if >= 0
$x->is_neg();   # true if <  0
$x->is_inf(sign);  # true if +inf, or -inf (default is '+')

$x->bcmp($y);   # compare numbers (undef,<0,=0,>0)
$x->bacmp($y);  # compare absolutely (undef,<0,=0,>0)
$x->sign();   # return the sign, either +,- or NaN
$x->digit($n);  # return the nth digit, counting from right
$x->digit(-$n);  # return the nth digit, counting from left

# The following all modify their first argument. If you want to preserve
# $x, use $z = $x->copy()->bXXX($y); See under L<CAVEATS> for why this is
# neccessary when mixing $a = $b assigments with non-overloaded math.

# set
$x->bzero();   # set $i to 0
$x->bnan();   # set $i to NaN
$x->bone();                  # set $x to +1
$x->bone('-');               # set $x to -1
$x->binf();                  # set $x to inf
$x->binf('-');               # set $x to -inf

$x->bneg();   # negation
$x->babs();   # absolute value
$x->bnorm();   # normalize (no-op)
$x->bnot();   # two's complement (bit wise not)
$x->binc();   # increment x by 1
$x->bdec();   # decrement x by 1

$x->badd($y);   # addition (add $y to $x)
$x->bsub($y);   # subtraction (subtract $y from $x)
```

```
$x->bmul($y);   # multiplication (multiply $x by $y)
$x->bdiv($y);   # divide, set $x to quotient
  # return (quo,rem) or quo if scalar

$x->bmod($y);   # modulus ($x % $y)
$x->bpow($y);   # power of arguments ($x ** $y)
$x->blsft($y);  # left shift
$x->brsft($y);  # right shift
  # return (quo,rem) or quo if scalar

$x->blog();   # logarithm of $x to base e (Euler's number)
$x->blog($base);  # logarithm of $x to base $base (f.i. 2)

$x->band($y);   # bit-wise and
$x->bior($y);   # bit-wise inclusive or
$x->bxor($y);   # bit-wise exclusive or
$x->bnot();   # bit-wise not (two's complement)

$x->bsqrt();   # calculate square-root
$x->broot($y);  # $y'th root of $x (e.g. $y == 3 => cubic root)
$x->bfac();   # factorial of $x (1*2*3*4*..$x)

$x->bround($N);   # accuracy: preserve $N digits
$x->bfround($N);  # precision: round to the $Nth digit

$x->bfloor();   # return integer less or equal than $x
$x->bceil();   # return integer greater or equal than $x

# The following do not modify their arguments:

bgcd(@values);  # greatest common divisor
blcm(@values);  # lowest common multiplicator

$x->bstr();   # return string
$x->bsstr();   # return string in scientific notation

$x->as_int();   # return $x as BigInt
$x->exponent();  # return exponent as BigInt
$x->mantissa();  # return mantissa as BigInt
$x->parts();   # return (mantissa,exponent) as BigInt

$x->length();   # number of digits (w/o sign and '.')
($l,$f) = $x->length(); # number of digits, and length of fraction

$x->precision();  # return P of $x (or global, if P of $x undef)
$x->precision($n);  # set P of $x to $n
$x->accuracy();  # return A of $x (or global, if A of $x undef)
$x->accuracy($n);  # set A $x to $n

# these get/set the appropriate global value for all BigFloat objects
Math::BigFloat->precision(); # Precision
Math::BigFloat->accuracy(); # Accuracy
```

```
Math::BigFloat->round_mode(); # rounding mode
```

DESCRIPTION

All operators (inlcuding basic math operations) are overloaded if you declare your big floating point numbers as

```
$i = new Math::BigFloat '12_3.456_789_123_456_789E-2';
```

Operations with overloaded operators preserve the arguments, which is exactly what you expect.

Canonical notation

Input to these routines are either BigFloat objects, or strings of the following four forms:

- `/^[+-]\d+$/`
- `/^[+-]\d+\.\d*$/`
- `/^[+-]\d+E[+-]?\d+$/`
- `/^[+-]\d*\.\d+E[+-]?\d+$/`

all with optional leading and trailing zeros and/or spaces. Additonally, numbers are allowed to have an underscore between any two digits.

Empty strings as well as other illegal numbers results in 'NaN'.

bnorm() on a BigFloat object is now effectively a no-op, since the numbers are always stored in normalized form. On a string, it creates a BigFloat object.

Output

Output values are BigFloat objects (normalized), except for bstr() and bsstr().

The string output will always have leading and trailing zeros stripped and drop a plus sign. `bstr()` will give you always the form with a decimal point, while `bsstr()` (s for scientific) gives you the scientific notation.

```
Input   bstr()  bsstr()
'-0'    '0'   '0E1'
   '  -123 123 123' '-123123123' '-123123123E0'
'00.0123'  '0.0123' '123E-4'
'123.45E-2'  '1.2345' '12345E-4'
'10E+3'    '10000'   '1E4'
```

Some routines (`is_odd()`, `is_even()`, `is_zero()`, `is_one()`, `is_nan()`) return true or false, while others (`bcmp()`, `bacmp()`) return either undef, <0, 0 or >0 and are suited for sort.

Actual math is done by using the class defined with `with` = Class;> (which defaults to BigInts) to represent the mantissa and exponent.

The sign `/^[+-]$/` is stored separately. The string 'NaN' is used to represent the result when input arguments are not numbers, as well as the result of dividing by zero.

mantissa(), exponent() and parts()

`mantissa()` and `exponent()` return the said parts of the BigFloat as BigInts such that:

```
$m = $x->mantissa();
$e = $x->exponent();
$y = $m * ( 10 ** $e );
print "ok\n" if $x == $y;
```

`($m,$e) = $x->parts();` is just a shortcut giving you both of them.

A zero is represented and returned as `0E1`, **not** `0E0` (after Knuth).

Currently the mantissa is reduced as much as possible, favouring higher exponents over lower ones (e.g. returning 1e7 instead of 10e6 or 10000000e0). This might change in the future, so do not depend on it.

Accuracy vs. Precision

See also: *Rounding*.

Math::BigFloat supports both precision (rounding to a certain place before or after the dot) and accuracy (rounding to a certain number of digits). For a full documentation, examples and tips on these topics please see the large section about rounding in *Math::BigInt*.

Since things like `sqrt(2)` or `1 / 3` must presented with a limited accuracy lest a operation consumes all resources, each operation produces no more than the requested number of digits.

If there is no gloabl precision or accuracy set, **and** the operation in question was not called with a requested precision or accuracy, **and** the input $x has no accuracy or precision set, then a fallback parameter will be used. For historical reasons, it is called `div_scale` and can be accessed via:

```
$d = Math::BigFloat->div_scale();  # query
Math::BigFloat->div_scale($n);   # set to $n digits
```

The default value for `div_scale` is 40.

In case the result of one operation has more digits than specified, it is rounded. The rounding mode taken is either the default mode, or the one supplied to the operation after the *scale*:

```
$x = Math::BigFloat->new(2);
Math::BigFloat->accuracy(5);  # 5 digits max
$y = $x->copy()->bdiv(3);  # will give 0.66667
$y = $x->copy()->bdiv(3,6);  # will give 0.666667
$y = $x->copy()->bdiv(3,6,undef,'odd'); # will give 0.666667
Math::BigFloat->round_mode('zero');
$y = $x->copy()->bdiv(3,6);  # will also give 0.666667
```

Note that `Math::BigFloat->accuracy()` and `Math::BigFloat->precision()` set the global variables, and thus **any** newly created number will be subject to the global rounding **immidiately**. This means that in the examples above, the 3 as argument to `bdiv()` will also get an accuracy of **5**.

It is less confusing to either calculate the result fully, and afterwards round it explicitely, or use the additional parameters to the math functions like so:

```
use Math::BigFloat;
$x = Math::BigFloat->new(2);
$y = $x->copy()->bdiv(3);
print $y->bround(5),"\n";  # will give 0.66667

or

use Math::BigFloat;
$x = Math::BigFloat->new(2);
$y = $x->copy()->bdiv(3,5);  # will give 0.66667
print "$y\n";
```

Rounding

ffround (+$scale)

Rounds to the $scale'th place left from the '.', counting from the dot. The first digit is numbered 1.

ffround (-$scale)

Rounds to the $scale'th place right from the '.', counting from the dot.

ffround (0)

Rounds to an integer.

fround (+$scale)

Preserves accuracy to $scale digits from the left (aka significant digits) and pads the rest with zeros. If the number is between 1 and -1, the significant digits count from the first non-zero after the '.'

fround (-$scale) and fround (0)

These are effectively no-ops.

All rounding functions take as a second parameter a rounding mode from one of the following: 'even', 'odd', '+inf', '-inf', 'zero' or 'trunc'.

The default rounding mode is 'even'. By using `Math::BigFloat->round_mode($round_mode);` you can get and set the default mode for subsequent rounding. The usage of `$Math::BigFloat::$round_mode` is no longer supported. The second parameter to the round functions then overrides the default temporarily.

The `as_number()` function returns a BigInt from a Math::BigFloat. It uses 'trunc' as rounding mode to make it equivalent to:

```
$x = 2.5;
$y = int($x) + 2;
```

You can override this by passing the desired rounding mode as parameter to `as_number()`:

```
$x = Math::BigFloat->new(2.5);
$y = $x->as_number('odd'); # $y = 3
```

METHODS

accuracy

```
$x->accuracy(5);                # local for $x
CLASS->accuracy(5);             # global for all members of CLASS
                                # Note: This also applies to new()!

$A = $x->accuracy();            # read out accuracy that affects $x
$A = CLASS->accuracy();         # read out global accuracy
```

Set or get the global or local accuracy, aka how many significant digits the results have. If you set a global accuracy, then this also applies to new()!

Warning! The accuracy *sticks*, e.g. once you created a number under the influence of `CLASS->accuracy($A)`, all results from math operations with that number will also be rounded.

In most cases, you should probably round the results explicitely using one of *round()*, *bround()* or *bfround()* or by passing the desired accuracy to the math operation as additional parameter:

```
my $x = Math::BigInt->new(30000);
```

```
        my $y = Math::BigInt->new(7);
        print scalar $x->copy()->bdiv($y, 2);           # print 4300
        print scalar $x->copy()->bdiv($y)->bround(2);   # print 4300
```

precision()

```
        $x->precision(-2);      # local for $x, round at the second digit
right of the dot
        $x->precision(2);       # ditto, round at the second digit left of
the dot

        CLASS->precision(5);    # Global for all members of CLASS
                                # This also applies to new()!
        CLASS->precision(-5);   # ditto

        $P = CLASS->precision();        # read out global precision
        $P = $x->precision();           # read out precision that affects
$x
```

Note: You probably want to use *accuracy()* instead. With *accuracy* you set the number of digits each result should have, with *precision* you set the place where to round!

Autocreating constants

After `use Math::BigFloat ':constant'` all the floating point constants in the given scope are converted to `Math::BigFloat`. This conversion happens at compile time.

In particular

```
  perl -MMath::BigFloat=:constant -e 'print 2E-100,"\n"'
```

prints the value of `2E-100`. Note that without conversion of constants the expression 2E-100 will be calculated as normal floating point number.

Please note that ':constant' does not affect integer constants, nor binary nor hexadecimal constants. Use *bignum* or *Math::BigInt* to get this to work.

Math library

Math with the numbers is done (by default) by a module called Math::BigInt::Calc. This is equivalent to saying:

```
 use Math::BigFloat lib => 'Calc';
```

You can change this by using:

```
 use Math::BigFloat lib => 'BitVect';
```

The following would first try to find Math::BigInt::Foo, then Math::BigInt::Bar, and when this also fails, revert to Math::BigInt::Calc:

```
 use Math::BigFloat lib => 'Foo,Math::BigInt::Bar';
```

Calc.pm uses as internal format an array of elements of some decimal base (usually 1e7, but this might be differen for some systems) with the least significant digit first, while BitVect.pm uses a bit vector of base 2, most significant bit first. Other modules might use even different means of representing the numbers. See the respective module documentation for further details.

Please note that Math::BigFloat does **not** use the denoted library itself, but it merely passes the lib argument to Math::BigInt. So, instead of the need to do:

```
use Math::BigInt lib => 'GMP';
use Math::BigFloat;
```

you can roll it all into one line:

```
use Math::BigFloat lib => 'GMP';
```

It is also possible to just require Math::BigFloat:

```
require Math::BigFloat;
```

This will load the neccessary things (like BigInt) when they are needed, and automatically.

Use the lib, Luke! And see *Using Math::BigInt::Lite* for more details than you ever wanted to know about loading a different library.

Using Math::BigInt::Lite

It is possible to use *Math::BigInt::Lite* with Math::BigFloat:

```
# 1
use Math::BigFloat with => 'Math::BigInt::Lite';
```

There is no need to "use Math::BigInt" or "use Math::BigInt::Lite", but you can combine these if you want. For instance, you may want to use Math::BigInt objects in your main script, too.

```
# 2
use Math::BigInt;
use Math::BigFloat with => 'Math::BigInt::Lite';
```

Of course, you can combine this with the `lib` parameter.

```
# 3
use Math::BigFloat with => 'Math::BigInt::Lite', lib => 'GMP,Pari';
```

There is no need for a "use Math::BigInt;" statement, even if you want to use Math::BigInt's, since Math::BigFloat will needs Math::BigInt and thus always loads it. But if you add it, add it **before**:

```
# 4
use Math::BigInt;
use Math::BigFloat with => 'Math::BigInt::Lite', lib => 'GMP,Pari';
```

Notice that the module with the last `lib` will "win" and thus it's lib will be used if the lib is available:

```
# 5
use Math::BigInt lib => 'Bar,Baz';
use Math::BigFloat with => 'Math::BigInt::Lite', lib => 'Foo';
```

That would try to load Foo, Bar, Baz and Calc (in that order). Or in other words, Math::BigFloat will try to retain previously loaded libs when you don't specify it onem but if you specify one, it will try to load them.

Actually, the lib loading order would be "Bar,Baz,Calc", and then "Foo,Bar,Baz,Calc", but independend of which lib exists, the result is the same as trying the latter load alone, except for the fact that one of Bar or Baz might be loaded needlessly in an intermidiate step (and thus hang around

and waste memory). If neither Bar nor Baz exist (or don't work/compile), they will still be tried to be loaded, but this is not as time/memory consuming as actually loading one of them. Still, this type of usage is not recommended due to these issues.

The old way (loading the lib only in BigInt) still works though:

```
# 6
use Math::BigInt lib => 'Bar,Baz';
use Math::BigFloat;
```

You can even load Math::BigInt afterwards:

```
# 7
use Math::BigFloat;
use Math::BigInt lib => 'Bar,Baz';
```

But this has the same problems like #5, it will first load Calc (Math::BigFloat needs Math::BigInt and thus loads it) and then later Bar or Baz, depending on which of them works and is usable/loadable. Since this loads Calc unnecc., it is not recommended.

Since it also possible to just require Math::BigFloat, this poses the question about what libary this will use:

```
require Math::BigFloat;
my $x = Math::BigFloat->new(123); $x += 123;
```

It will use Calc. Please note that the call to import() is still done, but only when you use for the first time some Math::BigFloat math (it is triggered via any constructor, so the first time you create a Math::BigFloat, the load will happen in the background). This means:

```
require Math::BigFloat;
Math::BigFloat->import ( lib => 'Foo,Bar' );
```

would be the same as:

```
use Math::BigFloat lib => 'Foo, Bar';
```

But don't try to be clever to insert some operations in between:

```
require Math::BigFloat;
my $x = Math::BigFloat->bone() + 4;  # load BigInt and Calc
Math::BigFloat->import( lib => 'Pari' ); # load Pari, too
$x = Math::BigFloat->bone()+4;   # now use Pari
```

While this works, it loads Calc needlessly. But maybe you just wanted that?

Examples #3 is highly recommended for daily usage.

BUGS

Please see the file BUGS in the CPAN distribution Math::BigInt for known bugs.

CAVEATS

stringify, bstr()

Both stringify and bstr() now drop the leading '+'. The old code would return '+1.23', the new returns '1.23'. See the documentation in *Math::BigInt* for reasoning and details.

bdiv

The following will probably not do what you expect:

```
print $c->bdiv(123.456),"\n";
```

It prints both quotient and reminder since print works in list context. Also, bdiv() will modify $c, so be carefull. You probably want to use

```
print $c / 123.456,"\n";
print scalar $c->bdiv(123.456),"\n";  # or if you want to modify $c
```

instead.

Modifying and =

Beware of:

```
$x = Math::BigFloat->new(5);
$y = $x;
```

It will not do what you think, e.g. making a copy of $x. Instead it just makes a second reference to the **same** object and stores it in $y. Thus anything that modifies $x will modify $y (except overloaded math operators), and vice versa. See *Math::BigInt* for details and how to avoid that.

bpow

`bpow()` now modifies the first argument, unlike the old code which left it alone and only returned the result. This is to be consistent with `badd()` etc. The first will modify $x, the second one won't:

```
print bpow($x,$i),"\n";  # modify $x
print $x->bpow($i),"\n";  # ditto
print $x ** $i,"\n";  # leave $x alone
```

precision() vs. accuracy()

A common pitfall is to use *precision()* when you want to round a result to a certain number of digits:

```
use Math::BigFloat;

Math::BigFloat->precision(4);  # does not do what you think it does
my $x = Math::BigFloat->new(12345); # rounds $x to "12000"!
print "$x\n";    # print "12000"
my $y = Math::BigFloat->new(3);  # rounds $y to "0"!
print "$y\n";    # print "0"
$z = $x / $y;    # 12000 / 0 => NaN!
print "$z\n";
print $z->precision(),"\n";  # 4
```

Replacing *precision* with *accuracy* is probably not what you want, either:

```
use Math::BigFloat;

Math::BigFloat->accuracy(4);  # enables global rounding:
my $x = Math::BigFloat->new(123456); # rounded immidiately to "12350"
print "$x\n";    # print "123500"
my $y = Math::BigFloat->new(3);  # rounded to "3
print "$y\n";    # print "3"
print $z = $x->copy()->bdiv($y),"\n"; # 41170
print $z->accuracy(),"\n";  # 4
```

What you want to use instead is:

```
use Math::BigFloat;
```

```
my $x = Math::BigFloat->new(123456); # no rounding
print "$x\n";    # print "123456"
my $y = Math::BigFloat->new(3);  # no rounding
print "$y\n";    # print "3"
print $z = $x->copy()->bdiv($y,4),"\n"; # 41150
print $z->accuracy(),"\n";  # undef
```

In addition to computing what you expected, the last example also does **not** "taint" the result with an accuracy or precision setting, which would influence any further operation.

SEE ALSO

Math::BigInt, *Math::BigRat* and *Math::Big* as well as *Math::BigInt::BitVect*, *Math::BigInt::Pari* and *Math::BigInt::GMP*.

The pragmas *bignum*, *bigint* and *bigrat* might also be of interest because they solve the autoupgrading/downgrading issue, at least partly.

The package at *http://search.cpan.org/search?mode=module&query=Math%3A%3ABigInt* contains more documentation including a full version history, testcases, empty subclass files and benchmarks.

LICENSE

AUTHORS

Mark Biggar, overloaded interface by Ilya Zakharevich. Completely rewritten by Tels *http://bloodgate.com* in 2001 - 2004, and still at it in 2005.